C/C++ Contributing Editors

The New C: Variable Length Arrays, Part 3: Pointers and Parameters

Randy Meyers

Randy shows that pointers to VLAs are as convenient as pointers to normal arrays. The question is: how many programmers understand pointers to arrays in the first place? Look here to see if you're one of them.

Every C programmer knows that pointers and arrays are closely related. In fact, many students learning C wonder how they differ once they are told that you can apply the square bracket indexing operator to both arrays and pointers, and that an array name becomes a pointer to the first element of the array except when the array name is the operand of **sizeof** or the address of operator (**unary &**).

Pointer arithmetic is one of the reasons why arrays and pointers are intertwined [1]; another reason is that many operations on arrays cannot be performed on arrays directly. In particular, you cannot pass an entire array as an argument to a function. Instead, a pointer to the array is passed, and the function operates on the array indirectly through the pointer. So close is the relationship between arrays and pointers that C syntax and semantics somewhat obscure the fact that C lacks array parameters.

It should not be surprising that the new VLA (variable length array) feature in C99 [2] has the companion feature of pointer to variable array, and that one of the useful places to use a pointer to a VLA is as a parameter to a function.

Pointer to VLA

A pointer to VLA can be declared using the syntax similar to pointer to (normal) array:

```
int (*pa)[10];
int (*pvla)[f()];
```

pa is a pointer to an array of 10 **int**s. **pvla** is a pointer to a VLA of the number of **int**s given by expression **f()** when the declaration is reached in the normal flow of control in the program. The difference between a pointer to an array and a pointer to a VLA is that the bounds of the (normal) array is a constant expression [3] while the bounds of the VLA is a run-time expression. Normally such a small difference between a new language feature and an old feature would mean that programmers would have little trouble understanding the new feature. Unfortunately, even though pointers to arrays date back to early C, many programmers are unfamiliar with them.

There are two reasons for this unfamiliarity. First, as we will see in the next section, pointers to arrays most naturally occur as function parameters, and C syntax and semantics handle this with so much grace that many programmers fail to notice.

Second, while pointers to arrays can be used to process a single dimensional array, it is more natural in C to process such an array using a pointer to the element type.

Consider Listing 1, which initializes the elements of an array and a VLA to **1** indirectly through pointers. The pattern in this code should look familiar to even a programmer with little experience with pointers to arrays. A pointer is declared. The pointer is assigned or initialized with a pointer to the object that it is to operate on, usually by applying the **&** operator to the object to be accessed indirectly. The ***** operator is applied to the pointer, and the result of the ***** operator is treated as if it was the original object referenced by the pointer. The only unusual thing about Listing 1 is that applying ***** to the pointers yields arrays.

Listing 2 is the more common way to write the function in Listing 1. In Listing 2, pointers to **int** are used to process the arrays of **int**s rather than using pointers to (normal or variable length) arrays of **int**s. In C, because of pointer arithmetic and the fact that the index operator is defined in terms of pointer arithmetic [1], a pointer of type **T** can also be used to process an array of type **T**. As Listing 2 shows, a pointer can process all of the elements of an array merely by indexing the pointer. (Remember, **E[i]** means ***(E + i)** regardless of whether **E** is an array or a pointer expression.)

Contrast the initializations of the pointers in Listings 1 and 2. In Listing 1, the initializations of **pa** and **pvla** use the **&** operator on arrays yielding respectively a pointer to an array of three **int**s and a pointer to a VLA of **bounds int**s. In Listing 2, the initializations of **p1** and **p2** just use the array names without the **&** operator. Whenever an array name appears in an expression except as the operand of **unary &** or the **sizeof** operator, the value of the array name becomes a pointer to the first element of the array. More formally, if **A** is an expression with type array, except when the operand of **unary &** or **sizeof**, **A** has the value and type of **&((A)[0])**. Thus in Listing 2, **p1** and **p2** are initialized with pointers to **int**s. Note that a single dimensional VLA yields a pointer type that carries no hint that it came from a VLA.

Given Listings 1 and 2, why are pointers to arrays needed at all? The answer is that pointers to arrays are useful when processing multidimensional arrays. Consider Listing 3.

Listing 3 seems to be a cross between Listing 1 and Listing 2 for good reasons. Listing 1 uses pointers to arrays, as does Listing 3. Listing 2 shows how a pointer to type **T** can be used to process an array with elements of type **T**, as does Listing 3. The difference between Listing 2 and Listing 3 is that in Listing 3 type **T** is an array type rather than a basic type like **int**.

In Listing 3, the pointers are pointers to arrays (normal or variable length). When you dereference a pointer to an array, the result is an array (which might then become a pointer to its first element, as described above). When you add one to a pointer to an array, then you move the pointer to the next entire array that follows the one the pointer originally pointed to. When you index a pointer to an array, each index selects an array object. Thus in Listing 3, **pa[i]** or **pvla[i]** yields an array object that may be further indexed. As I wrote above, in C, if you have a pointer to type **T**, you can use it to process an array of type **T**, even if type **T** is an array type. Note that when **pa** and **pvla** are initialized in Listing 3 that just the array names are used (no **&** operator). As explained above, the array names become pointers to their first elements. Thus, **pa** is initialized with **&(a[0])**, a pointer to an array of three **int**s. **pvla** is initialized with **&(vla[0])**, a pointer to an array of **bounds int**s.

As I discussed in [1], pointer arithmetic in C requires knowing the size of the object that the pointer is pointing to. In Listing 3, the size of the objects pointed to by **pa** is known at compile time: it is **sizeof (int [3])**. In contrast, the size of the objects pointed to by **pvla** is not known at compile time: it is **sizeof(int [bounds])**. As I discussed in [2], the result of a **sizeof** operator is computed at run time for a VLA. Not surprisingly, **sizeof** is also a run-time operation if you ask for the size of a VLA reached indirectly through a pointer.

Thus in Listing 3, **sizeof (*pvla)** is an expression whose value is computed at run time and is equal to **sizeof(int [bounds])**. If **sizeof(int)** is **4** and **bounds** had the value **3**, the result of the **sizeof** expressions would be **12**. Note that **sizeof (pvla)** is not a run-time operation since it is just the size of the pointer **pvla** itself, which is known at compile time.

sizeof(*pvla) is used whenever pointer arithmetic or indexing is done on **pvla**. This means that the C implementation must record the size of the VLA type that the pointer type points to. Like other VLA types, the size information associated with a pointer to VLA is saved when the declaration is executed and does not change during the declaration's lifetime. The expression that is the **bounds** of the VLA is not reevaluated until next time the declaration is executed.

Consider Listing 4. (By the way, the "z" in the format is a new C99 modifier that means the argument is **size_t** or the corresponding signed integer type. Thus, "%zu" prints a **size_t** number as unsigned.) When run, the program in Listing 4 prints out **10 20 30** even though the value of **n** changes between when the pointer to VLA is declared and the **sizeof** expression that yields the size of the array pointed to. However, since each pass through the loop enters and exits the block that is the loop body, each pass of the loop picks up a new value of **n** for the **bounds** of the pointer to VLA. Listings 1, 2, and 3 show a useful coding technique. Although from the C implementation's point of view the **bounds** of a VLA are fixed from the time its declaration is executed until the lifetime of the declaration ends, that does not mean that the programmer can conveniently compute that **bounds** later in the program. If the **bounds** expression of a VLA is complex or might change value, you might want to assign the value of the **bounds** expression to a local variable and use the local variable as the **bounds** in the declaration. If you fail to do this, all is not lost: see the discussion of **sizeof** in [2].

Listing 4 also shows another point about the size information that the C implementation saves for VLAs. That size information is associated with the type and not the value of the pointer to VLA or even the VLA object itself. In Listing 4, **pvla** is uninitialized stack trash (that is OK since the **sizeof** expression does not actually evaluate its operand: **pvla** is never actually dereferenced). Clearly, the size of the array that **pvla** is suppose to point to is not part of the value of **pvla**. Likewise, there is no array to which **pvla** points in Listing 4, so the size is not part of the array object. Instead, every VLA type in a program causes the C implementation to set aside an unnamed variable to hold the size of arrays of that type. (The optimizer might combine several such variables into one if it proves that they hold the same value).

Note that this approach uses less memory than making the size information part of the array object itself. Consider the declaration:

```
int x[m][n];
```

C needs only storage to hold two sizes: the size of the two-dimensional array **x** and the size of the elements of **x** (all of which are the same size). If size was part of the array object (Java does things this way), the **x** object itself would contain space to store the entire size of the array, and each element of **x** would contain space to store the size of the subarray that is that element. The Java approach needs to store **m+1** lengths while the C approach stores only two. Another advantage of the C approach is since the size information is not part of the object, a VLA has the same size and storage layout as a non-VLA with the same element type and array **bounds**.

Since the size is not part of the array, and VLAs and normal arrays have the same representation, pointers to VLAs are permitted to point at normal arrays and pointers to normal arrays are permitted to point at VLAs. When you assign a pointer to a (normal or variable length) array to a pointer to a (normal or variable length) array, the number of elements and the type of the elements must match. As long as both pointers point to normal arrays, both of these requirements are checked at compile time. If either the source or destination of the assignment is a pointer to a VLA, then only the element types will be checked by the compiler. The number of elements must still match. If they do not, then the program has a run-time error that the C implementation has no obligation to catch: the program might terminate immediately or continue to run with mysterious behavior or even appear to work. Listing 5 shows the four combinations of assignments between pointers to (normal or variable length) arrays. All four pointer assignments are valid, as long as function **ex5** is called with the argument **10**. Only the first and last assignments are valid if any other argument is passed. Only the first assignment is checked at compile time for the proper number of elements in the array types.

VLA Parameters

C does not permit function parameters to be arrays; it does not permit entire arrays to be passed as arguments. However, the language finesses this so well that some programmers do not realize this.

If a parameter to a function has type array of type **T**, the C compiler rewrites the declaration of the parameter to be a pointer to type **T**. The result is exactly as if the programmer wrote the pointer form of the parameter declaration in the first place. This rule is followed whether the array parameter is a normal array or a VLA.

Note how the semantics of C dovetail to handle “array” parameters seamlessly. If you write an array parameter, the compiler rewrites the function declaration to have a pointer parameter. This is OK since you can write the body of the function as if the parameter was still an array since the index operator works as well on a pointer as on an array. When you call the function, you can pass an array name as an argument because any expression of type array (except when the operand of **unary &** or **sizeof**) becomes a pointer. If the array being passed as an argument was compatible with the original array type of parameter, then the pointer type of the argument will match the pointer rewrite of the parameter. This works the same for VLAs or normal arrays.

One somewhat surprising thing is that single-dimensional VLA parameters become plain old pointers after the rewrite. Consider:

```
void f(int n, int a[n])
```

after the compiler automatically rewrites the function, it becomes:

```
void f(int n, int *a)
```

However, multiple dimensional VLA parameters become pointers to VLAs after the rewrite. For example,

```
void g(int n, int a[n][n+1])
```

becomes:

```
void g(int n, int (*a)[n+1])
```

Of course, multiple dimensional normal array parameters become pointers to normal arrays. It is in this context that most C programmers have used pointers to arrays without realizing it.

The act of passing an "array" argument to an "array" function parameter is really a form of pointer assignment and works as described in the previous section. Thus you can pass either a normal array or a VLA to a function whose parameter is a normal array. You can also pass either a normal array or a VLA to a function whose parameter is a VLA.

Listing 6 shows a function that sets the diagonal of its square array parameter to one and sets all other elements to zero. This function can be called on any **n** by **n** array of **int**s since the **bounds** of the array is passed as an argument. It is fairly common for the **bounds** of VLA parameters to be another parameter to the same function as in Listing 6, but this is not required. The run-time expression that is the **bounds** of the VLA may be any expression involving any variables or functions that are in scope at the time the parameter is declared. The **bounds** expression is evaluated each time the function is called since calling the function causes its parameter's declarations to be executed, and the lifetime of the parameters ends when the function returns.

Function prototypes with VLA parameters can be written just like the function definition. For example, a prototype for the function in Listing 6 could be:

```
void diag(int n, int a[n][n]);
```

There is an advantage to writing the prototype that way since it makes clear the relationship between the parameter **n** and the **bounds** of **a**. However, the **bounds** expression is not really needed for the prototype, and sometimes the **bounds** expressions might be complex or reference identifiers only in scope at the point of the function definition. Because of this, the **bounds** expression of a VLA in a function prototype may be replaced with a "*" character. In this context, the asterisk is just a placeholder for a run-time expression that will appear in the function definition. Thus, the function prototype for the function in Listing 6 can also be written as:

```
void diag(int n, int a[*][*]);
```

As far as the compiler is concerned, the two prototypes for **diag** given above are identical.

Limitations on Pointers to VLAs

Like VLAs, pointers to VLAs cannot appear at file scope. They must be either parameters to function prototypes or local variables of a block. (The C Standard considers a function's parameters to be locals of the block that is the function body.)

Pointers to VLAs may not be **static** or **extern**. Such objects have a lifetime that starts before **main** is called and ends when the program exits. Since the size information for a VLA or pointer to VLA is fixed during its lifetime, such objects would have a size fixed during the running of the program. That sort of takes the variable out of variable length.

References

[1] Randy Meyers. "The New C: Why Variable Length Arrays?" *C/C++ Users Journal*, October 2001.

[2] Randy Meyers. "The New C: Variable Length Arrays, Part 2," *C/C++ Users Journal*, December 2001.

[3] Randy Meyers. "The New C: Declarations and Initializations," *C/C++ Users Journal*, April 2001, <www.cuj.com/reference/articles/2001/0104/0104d/0104d.htm>.

Randy Meyers *is a consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*